

COMPUTING SCIENCE

Impact of Policy Design on Workflow Resiliency Computation Time

John C. Mace, Charles Morisset and Aad van Moorsel

TECHNICAL REPORT SERIES

No. CS-TR-1469

May 2015

No. CS-TR-1469

May, 2015

Impact of Policy Design on Workflow Resiliency Computation Time

J. C. Mace, C. Morisset and A. Moorsel

Abstract

Workflows are complex operational processes that include security constraints restricting which users can perform which tasks. An improper user-task assignment may prevent the completion of the workflow, and deciding such an assignment at runtime is known to be complex, especially when considering user unavailability (known as the resiliency problem). Therefore, design tools are required that allow fast evaluation of workflow resiliency. In this paper, we propose a methodology for workflow designers to assess the impact of the security policy on computing the resiliency of a workflow. Our approach relies on encoding a workflow into the probabilistic model-checker PRISM, allowing its resiliency to be evaluated by solving a Markov Decision Process. We observe and illustrate that adding or removing some constraints has a clear impact on the resiliency computation time, and we compute the set of security constraints that can be artificially added to a security policy in order to reduce the computation time while maintaining the resiliency.

Bibliographical details

MACE, J. C; MORISSET, C; MOORSEL, A;
Impact of Policy Design on Workflow Resiliency Computation Time
[By] J. C. Mace, C. Morisset, A. Moorsel
Newcastle upon Tyne: Newcastle University: Computing Science, 2015.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1469)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1469

Abstract

Workflows are complex operational processes that include security constraints restricting which users can perform which tasks. An improper user-task assignment may prevent the completion of the workflow, and deciding such an assignment at runtime is known to be complex, especially when considering user unavailability (known as the resiliency problem). Therefore, design tools are required that allow fast evaluation of workflow resiliency. In this paper, we propose a methodology for workflow designers to assess the impact of the security policy on computing the resiliency of a workflow. Our approach relies on encoding a workflow into the probabilistic model-checker PRISM, allowing its resiliency to be evaluated by solving a Markov Decision Process. We observe and illustrate that adding or removing some constraints has a clear impact on the resiliency computation time, and we compute the set of security constraints that can be artificially added to a security policy in order to reduce the computation time while maintaining the resiliency.

About the authors

John Mace is a research assistant at Newcastle University, working with Charles Morisset and Aad van Moorsel on devising new tools and methodologies to analyse the impact of information security using, in particular security ontology development tools and more formal techniques to quantify security impact on workflows. John completed a BSc (Hons) in Computing Science at Newcastle University in 2010 during which time he received a Scott Logic prize for computer excellence for his year two performance. He was also awarded the School of Computing Science prize for best overall performance in year three. John is due to submit his PhD thesis mid-2015.

Charles Morisset is a Senior Research Associate at Newcastle University, working with Aad van Moorsel on quantitative aspects of security, in particular in the decision making process and in access control mechanisms. Charles received his PhD from Universite Pierre et Marie Curie - Paris VI in France in 2007, on the topic of formalisation of access control systems. He then worked from 2007 to 2009 at the United Nations University, in Macau SAR, China, on formal methods for software engineering, after which he joined the Information Security Group at Royal Holloway, University of London, to work on risk-based access control until 2011. From 2011 to 2013, he worked at the Istituto di Informatica e Telematica in Pisa, Italy, on formal methods and access control, and he joined the Centre for Cybercrime and Computer Security at Newcastle University in 2013.

Aad van Moorsel is a Professor in Distributed Systems and Head of School at the School of Computing Science in Newcastle University. His group conducts research in security, privacy and trust. Almost all of the group's research contains elements of quantification, be it through system measurement, predictive modelling or on-line adaptation. Aad worked in industry from 1996 until 2003, first as a researcher at Bell Labs/Lucent Technologies in Murray Hill and then as a research manager at Hewlett-Packard Labs in Palo Alto, both in the United States. He got his PhD in computer science from Universiteit Twente in The Netherlands (1993) and has a Masters in mathematics from Universiteit Leiden, also in The Netherlands. After finishing his PhD he was a postdoc at the University of Illinois at Urbana-Champaign, Illinois, USA, for two years. Aad became the Head of the School of Computing Science in 2012.

Suggested keywords

WORKFLOW SATISFIABILITY PROBLEM
PROBABILISTIC MODEL CHECKER
USER AVAILABILITY

Impact of Policy Design on Workflow Resiliency Computation Time

John C. Mace, Charles Morisset, and Aad van Moorsel

School of Computing Science,
Newcastle University, Newcastle upon Tyne,
NE1 7RU, United Kingdom
{j.c.mace,charles.morisset,aad.vanmoorsel}@ncl.ac.uk

Abstract. Workflows are complex operational processes that include security constraints restricting which users can perform which tasks. An improper user-task assignment may prevent the completion of the workflow, and deciding such an assignment at runtime is known to be complex, especially when considering user unavailability (known as the resiliency problem). Therefore, design tools are required that allow fast evaluation of workflow resiliency. In this paper, we propose a methodology for workflow designers to assess the impact of the security policy on computing the resiliency of a workflow. Our approach relies on encoding a workflow into the probabilistic model-checker PRISM, allowing its resiliency to be evaluated by solving a Markov Decision Process. We observe and illustrate that adding or removing some constraints has a clear impact on the resiliency computation time, and we compute the set of security constraints that can be artificially added to a security policy in order to reduce the computation time while maintaining the resiliency.

Keywords: Workflow Satisfiability Problem, Probabilistic Model Checker, User Availability

1 Introduction

Workflows are used in multiple domains, for instance business environments, to represent complex operational processes [7, 14], or healthcare environments, to represent the different protocols that must be respected [25]. There is also an increasing interest in scientific environments, where platforms like eScience Central [17] allow domain experts to define scientific processes, which are then automatically deployed and executed. Although the exact definition can change from one context to another, a workflow typically consists of a partially ordered set of tasks, where each task must be executed by a user [1]. Workflow designers may have to impose complex security policies, restricting which users can perform which tasks. This includes static user-task permissions but also dynamic constraints, such as separation or binding of duty constraints, which indicate tasks that cannot be performed by the same user [19], or tasks that must be performed by the same user [9], respectively.

In general, purely granting an assignment request for a task based on its user permissions and constraints with previously executed tasks may not be enough. Assigning a specific user to a task can prevent the completion of the workflow at a later stage, meaning in general, all possible options have to be considered. Checking that a particular user-task assignment is both valid and allows the workflow to finish is known as the workflow satisfiability problem (WSP) and has been shown to be NP-hard [12,27], indicating the runtime assignment process may be computationally demanding.

Workflow resiliency extends the WSP by considering users may become unavailable at runtime, a concept first introduced by Wang and Li [27]. This problem was later refined by Mace et al. [22], who considered a more quantitative approach, where each user is associated with a probability to become unavailable, and showed that calculating the resiliency of a workflow was equivalent to finding the optimal policy of a Markov Decision Process (MDP). The value returned by the value function of the MDP provides a measure of likely workflow completion. Therefore, evaluating resiliency at runtime can ensure assignments are granted only if the rest of the workflow can be satisfied with a probability above a given resiliency threshold.

Indeed, contrary to the WSP, which can be solved at design time, maximising the resiliency of workflow requires to re-compute at each step the expected resiliency, in order to adjust the user-task assignment to the current availability of the users. Hence, evaluating resiliency for assignments at runtime has itself an impact on workflow execution time. Recent optimising approaches for the WSP, such as [12], and algorithms and tools, such as model-checking [2], have been proposed, however, they are not directly concerned with user availability.

In this paper, we investigate how to improve the computation time for the resiliency of a workflow at runtime. In particular, we observe that adding or removing security components to the security policy has a clear impact on the resiliency computation time, that can be either increased or decreased. We therefore propose a methodology to help a workflow designer assess the impact of such policy changes. We apply this methodology to show how to compute the set of security constraints that can be added to a workflow, without impacting the actual resiliency while significantly decreasing the resiliency computation time.

After discussing the related work (Section 2) and formally defining the notion of workflow resiliency (Section 3), we present the contributions of this paper, which are: the automated analysis of workflow resiliency, using an encoding in the probabilistic model checker PRISM [20] of the theoretical approach presented in [22] (Section 4); the empirical assessment of policy changes on the resiliency computation time (Section 5); the methodology to calculate a set of *artificial* security policy constraints, in order to reduce the resiliency computation time while maintaining the actual resiliency value, and its illustration on an example (Section 6). We believe that building efficient tools for the analysis of workflows will be helpful to workflow designers, by helping them understanding the complexity of the workflow they are building, and estimating the potential runtime impact of their security policy designs.

2 Related Work

A number of previous studies on workflow resiliency appear in the literature. Wang and Li took a first step in [27] to quantify resiliency by addressing the problem of whether a workflow can still complete in the absence of users and defined a workflow as k resilient to all failures of up to k users across an entire workflow. Lowalekar et al. in [21] use security attributes to choose the most favourable between multiple assignments exhibiting the same level of k resiliency.

Basin et al. consider the impact of security on resiliency by allowing user-task permission changes to overcome user failure induced workflow blocks, at a quantifiable cost [5, 6]. Wainer et al. also consider in [26] the explicit overriding of security constraints in workflows, by defining a notion of privilege. Similarly, Bakkali [4] suggests overcoming user unavailability through selected delegation and the placement of criticality values over workflows.

A mechanism for the specification and enforcement of workflow authorisation constraints is given by Bertino et al. in [8] whilst Ayed et al. discuss security policy definition and deployment for workflow management systems in [3]. Model checking has been used by Armando et al. [2] to formally model and automatically analyse security constrained business processes to ensure they meet given security properties. He et al. in [15] also use modelling techniques to analyse security constraint impact in terms of computational time and resources on workflow execution.

Herbert et al. in [16] model workflows expressed in BPMN as MDPs. The probabilistic model checker PRISM is utilised to check various probabilistic properties such as reaching particular states of interest, or the occurrence and ordering of certain events. Calinescu et al. use PRISM to evaluate the Quality of Service (QoS) delivered by dynamically composed service-based systems [11]. PRISM has also been used for identifying and recovering from runtime requirement violations in dynamically adaptable application software [10]. Quantitative access control using partially-observable MDPs is presented by Martinelli et al. in [24] which under uncertainty, aims to optimise the decision process for a sequence of access requests.

However, to the best of our knowledge, there is no current literature neither on automatic analysis of workflow resiliency, nor on the analysis of how changes to a workflow's security policy impact resiliency computation, which is the focus of this paper.

3 Workflow

In this section we provide our working definition of a workflow and describe the process of assigning users to tasks whilst respecting the security policy, known as the workflow satisfiability problem (WSP). We then describe the notion of workflow resiliency which looks to solve the WSP under the assumption users may become unavailable for future task assignments.

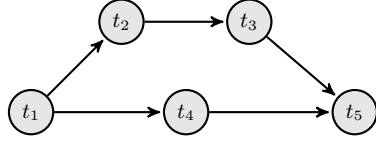


Fig. 1. Running example workflow

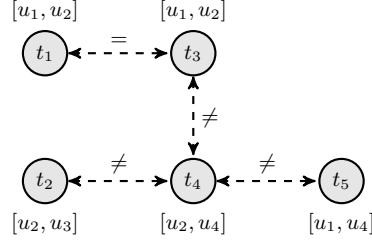


Fig. 2. Running example security policy

3.1 Workflow Definition

We define here a workflow in similar fashion to Wang and Li [27] and Crampton et al. [12]. A workflow firstly consists of a partially ordered set of tasks $(T, <)$, such that for any two tasks $t, t' \in T$, if $t < t'$, then t must be performed before t' in any given instance of the workflow.

Each task needs to be assigned to a user in a given set U , and such an assignment must respect a *security policy*. In general, a policy is a triple $p = (P, S, B)$ where:

- $P \subseteq U \times T$ is a set of *user-task permissions*, such that $(u, t) \in P$ if, and only if u is allowed to perform t .
- $S \subseteq T \times T$ is a set of *separations of duty*, such that $(t, t') \in S$ if, and only if the users assigned to t and t' are distinct.
- $B \subseteq T \times T$ is a set of *bindings of duty*, such that $(t, t') \in B$ if, and only if the same user is assigned to t and t' .

Definition 1 (Workflow). A workflow is a tuple $w = (U, (T, <), p)$, where U is a set of users, T is a partially ordered set of tasks, and p is a security policy.

Running example. As a running example to illustrate the different concepts presented here, we consider the workflow $w_1 = (U_1, (T_1, <), p_1)$, where $U_1 = \{u_1, u_2, u_3, u_4\}$, $T_1 = \{t_1, t_2, t_3, t_4, t_5\}$ such that $t_1 < t_2 < t_3 < t_5$ and $t_1 < t_4 < t_5$, and the p_1 is defined as the triple (P_1, S_1, B_1) where:

- $P_1 = \{(u_1, t_1), (u_2, t_1), (u_2, t_2), (u_3, t_2), (u_1, t_3), (u_2, t_3), (u_2, t_4), (u_4, t_4), (u_1, t_5), (u_4, t_5)\}$
- $S_1 = \{(t_2, t_4), (t_3, t_4), (t_4, t_5)\}$
- $B_1 = \{(t_1, t_3)\}$

Figure 1 illustrates the task ordering over T_1 and Figure 2 illustrates this security policy, where a dotted arrow signifies a constraint between the tasks t and t' labelled \neq to indicate a separation of duty, and $=$ to indicate a binding of duty. A label $[u_i, \dots, u_j]$ states the users that are authorised by P_1 to execute t .

3.2 Workflow Satisfiability Problem

A *workflow assignment* is a relation $A \subseteq U \times T$, such that $(u_i, t_i) \in A$ indicates that user u_i is assigned to the task t_i . Intuitively, A is *valid* when i) the

task ordering is respected; *ii*) all assignments are permitted by the user-task permission; *iii*) separation and binding constraints are respected; *iv*) no task is executed twice. More formally, given a workflow $w = (U, (T, <), (P, S, B))$, A is a valid assignment, and in this case we write $A \vdash w$ if, and only if the following five conditions are met:

$$\forall (u, t) \in A \quad \forall t' \in T \quad t' < t \Rightarrow \exists u' \in U \quad (u', t') \in A \quad (1)$$

$$A \subseteq P \quad (2)$$

$$\forall (t, t') \in S \quad \exists (u, t) \in A \quad \exists (u', t') \in A \Rightarrow u' \neq u \quad (3)$$

$$\forall (t, t') \in B \quad \exists (u, t) \in A \quad \exists (u', t') \in A \Rightarrow u' = u \quad (4)$$

$$\forall t \in T \quad \forall u, u' \in U \quad (u, t) \in A \wedge (u', t) \in A \Rightarrow u = u' \quad (5)$$

A workflow assignment A is said to be a *partial* if it does not include an assignment for every task in the workflow. For instance, in our running example, $\{(u_1, t_1), (u_3, t_2), (u_2, t_4)\}$ is a valid partial assignment whereas $\{(u_1, t_1), (u_2, t_2), (u_2, t_4)\}$ is not as it violates the separation of duty constraint between tasks t_2 and t_4 . For a workflow to complete successfully, *every* task needs to be assigned a user for execution. A workflow assignment A is therefore said to be *complete*, if, and only if:

$$\forall t \in T \quad \exists u \in U \quad (u, t) \in A \quad (6)$$

The workflow satisfiability problem (WSP) consists of finding a complete and valid assignment, and in some cases can be relatively simple. For instance, consider a policy where $S = B = \emptyset$, i.e., where there are no separations or bindings of duty. In this case, it is enough to assign each task t with a user u such that $(u, t) \in P$. If there is no such user, the workflow is unsatisfiable. However, in general, the WSP has been shown to be NP hard [27], i.e., roughly speaking, finding a complete and valid assignment might require to check all possible assignments. With our running example, imagine we want to find a complete assignment for w_1 and begin assigning users to tasks t_1, t_2 and t_4 to form the partial assignment $A = \{(u_2, t_1), (u_3, t_2), (u_2, t_4)\}$. Although this assignment is valid, there is no user u such that $A \cup \{(u, t_3)\}$ is also valid, meaning that the workflow cannot finish. However with the partial assignment $\{(u_2, t_1), (u_3, t_2), (u_4, t_4)\}$, we can add (u_2, t_3) and (u_1, t_5) to form a valid and complete assignment.

3.3 Workflow Resiliency

Solving the WSP assumes users will always be available for future tasks, however in practice, sickness, vacation, heavy workloads, etc., can cause users to be unavailable for a given user-task assignment. It is important to take this into account when finding A for a given workflow. This is called the resiliency problem, whether a workflow can be satisfied even when some users become absent.

Wang and Li defined an approach to calculate a valid assignment if one exists, that is resilient to up to k users failing, in other words declaring a workflow to be either k resilient or not [27]. This approach is rather *binary* as in many cases,

finding an assignment for a workflow that is resilient to every combination of k user failures may be impossible. Yet finding a valid assignment that is resilient in 9 out of 10 cases is arguably better than choosing a valid assignment that is resilient in only 1 out of 10 cases.

The problem of resiliency adds another level of complexity to the WSP. For instance, consider $\{(u_1, t_1), (u_2, t_2), (u_1, t_3), (u_4, t_4), (u_1, t_5)\} \vdash w_1$ in our running example, where u_4 has a very high probability of failing at or before t_4 . If u_4 does fail, t_4 cannot be reassigned to any other user meaning the workflow cannot finish. If we chose a different assignment $\{(u_1, t_1), (u_3, t_2), (u_1, t_3), (u_4, t_4), (u_1, t_5)\} \vdash w_1$, intuitively the workflow is more resilient as t_4 can be reassigned to u_2 and still finish if u_4 did indeed fail. In [22], Mace et al introduce probabilistic user failures and show that computing the optimal policy of an MDP is equivalent to finding $A \vdash w$ that maximises the value function. The value function returns $0 < v \leq 1$ if there exists $A \vdash w$ where v indicates the probability of the workflow to finish, or 0 otherwise.

Moreover, Mace et al define in [23] several user availability models and discuss the effects model choice can have on workflow resiliency analysis. In this paper we consider a dynamic user availability model meaning any user who becomes unavailable for a task may become available again at any step later in the workflow.

4 Computing Workflow Resiliency at Runtime

Although user availability is modelled in a probabilistic way, at runtime, a user is either available or not. In other words, the resiliency of a workflow denotes a prediction of completion, and not a level a completion: a workflow only terminates if all tasks have been assigned to a user available for that task. When the availability of users does not change at runtime, any valid assignment computed before execution remains valid throughout execution. However, when user availability is dynamic, the validity of an assignment might change during the execution, and therefore a new assignment might need to be found.

According to Crampton and Khambhammettu [13], there are two main workflow execution models: *workflow-driven execution model* (WDEM), where users are automatically assigned tasks to execute, and *user-driven execution model* (UDEM), where users initiate requests to be assigned tasks at runtime. The impact of dynamic user availability is slightly different between the two models: with WDEM, intuitively, we want to continuously compute the most resilient assignment, adapting to changes in user availability; Whereas with UDEM, we want to ensure that a user asking to execute a specific task either belongs to the most resilient assignment, or satisfies a threshold of resiliency.

With either model, resiliency might then need to be recomputed at runtime, which can be done by solving a Markov Decision Process (MDP) [22]. There are many ways to solve an MDP including dynamic programming (e.g. value iteration) [18]. This technique is provided by the probabilistic model checking tool PRISM, which enables the specification, construction and analysis of prob-

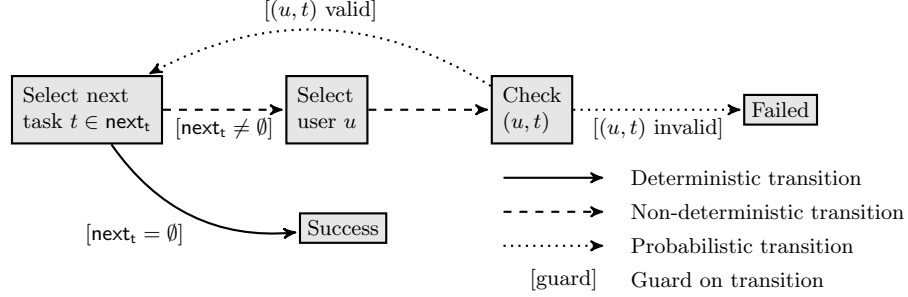


Fig. 3. Process computing the resiliency of a workflow, where next_t denotes the set of tasks remaining to be executed.

abilistic models such as MDPs [20]. PRISM is an intuitive choice as it can model both probabilistic and non-deterministic choice, and gives an efficient way to solve an MDP whilst providing analysis data regarding computation overheads. An overview of the PRISM modelling language is given in Appendix A.1.

The workflow assignment process is shown as a state diagram in Figure 3. Each node represents a process state while each directed arrow between two states s and s' indicates a transition from state s to state s' . The assignment process works as follows: 1) given a starting state the next unassigned task t in the workflow is selected, where the set of possible tasks is represented by next_t . Task selection is in general non-deterministic since several tasks can be the next one (i.e. in the case of parallel execution). If all tasks have been assigned then $\text{next}_t = \emptyset$; 2) When a task t is selected, an arbitrary user u is selected to be assigned to t . The selection of u is non-deterministic as the MDP will essentially try every user for each task assignment. 3) The user-task assignment (u, t) is then checked to see whether it is valid; in other words whether u available and (u, t) satisfies the workflow’s security policy p . This check is probabilistic, since user availability is probabilistic. If (u, t) is valid, u is assigned to t and the process starts again with the next task, otherwise the workflow terminates early.

The resiliency of the workflow is therefore computed as the maximal probability of reaching the state **Success**. We provide a full PRISM encoding of our running example in Appendix A.2. In our running example the resiliency is computed to be 51.16% with the probabilistic user availabilities given in Table 4, Appendix A.4.

5 Empirical Assessment of Policy Changes

In this section we provide an empirical assessment of resiliency computation time to help understanding of how it can be improved at runtime. In doing so we investigate the impact upon computation time of adding security constraints to the security policy.

Table 1. Result averages when applying randomly generated security policies to a workflow with 10 tasks and 5 users

	0	1~5	6~10	11~15	16~20	21~25	26~30	31~35	36~40	41~45
Resiliency (%)	58.23	57.97	55.73	52.78	50.49	46.02	34.85	15.31	0.89	0
0% resiliency	0	0	0	1	0	11	90	305	488	500
Computation (s)	0.11	0.38	1.56	2.24	1.80	1.08	0.52	0.20	0.07	0.04
Build time (s)	0.56	2.83	16.12	25.91	21.72	13.81	7.52	4.38	2.55	1.78
Total time (s)	0.67	3.21	17.68	28.15	23.52	14.89	8.04	4.58	2.62	1.82
States	3893	58246	346992	600287	522850	332259	171627	89361	47140	29387
Transitions	73249	758351	3352889	4754705	3649065	2171394	1090709	561534	294596	182751

5.1 Assessment Methodology

We first consider one workflow with 10 tasks and 5 users. For simplicity we only consider the addition of separation of duty constraints which is sufficient to show the changes to resiliency computation time. The maximum number of separation of duty constraints for a workflow of 10 tasks is 45 constraints. For each i where $0 \leq i \leq 45$ we generate 100 random security policies such that the permissions policy P contains between 2 and 5 users for each task, the separations of duty policy S contains i constraints, and the bindings of duty policy B has 0 constraints. Each policy is applied to the workflow meaning in all we analyse 4500 workflows using a computing platform incorporating a 2.40Ghz i7-4500U Intel processor and 8GB RAM. To take into account any influence the computing platform may have on analysis time, each analysis is repeated 50 times for each workflow and the average values taken.

In terms of user availability we use a dynamic availability model with probabilities of between 0.8 and 1.0 for each user u to be available for a task t (Table 5, Appendix A.4). The resiliency of each workflow is calculated with an unmodified version 4.2.1 of the PRISM model checker using the *explicit* engine. This is suitable for models with a potentially very large state space, only a fraction of which is actually reachable. A test program has been implemented which, given a number of inputs (number of workflows, tasks, users, etc) creates the required workflows with randomly generated security policies and generates the corresponding PRISM definition files. Each file is passed in turn to the PRISM model checker which logs the output composed of the resiliency value and other computational values including computation time.

5.2 Results

The results shown in Table 1 are given for our workflow with random separation of duty constraints applied, from 0 to 45. To place the workflow resiliency value and its computation time into perspective the following result averages are provided:

- **Resiliency** : workflow resiliency value
- **0% resiliency** : number of workflows unable to complete
- **Computation** : time to verify the state **Success** is reachable (Section 4)

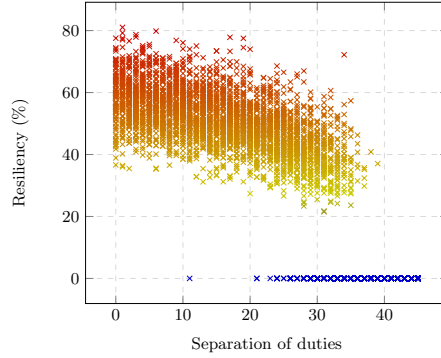


Fig. 4. Resiliency values for a workflow with 10 tasks and 5 users

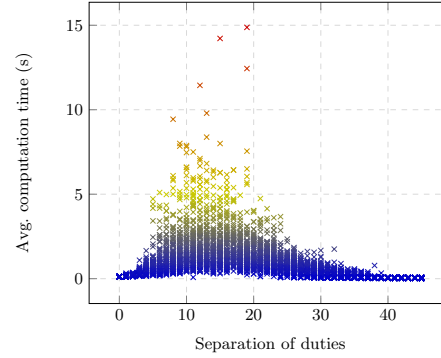


Fig. 5. Average computation times for a workflow with 10 tasks and 5 users

- **Build time** : time to build PRISM model
- **Total time** : computation + build time
- **States** : number of reachable states in PRISM model
- **Transitions** : number of transitions between states

The following sections provide more detailed analysis of resiliency and computation time.

Resiliency Analysis Note the resiliency returned by a single execution of a workflow is the same as the average resiliency of 50 executions, in other words the resiliency does not change due to fixed parameters. Each plot in Figure 4 therefore represents a set of workflows with the same resiliency value and the same number of separation of duty constraints (although each workflow in the set comes with a different set of constraints).

Figure 4 shows how in general, resiliency steadily reduces following an incremental introduction of separation of duty constraints. For example, with no constraints, the workflows generally have between 40 and 80% resiliency and with 20 constraints between 30 and 70%. All the workflows with between 0 and 10 constraints are shown to be resilient to some degree and up to the point where 20 constraints are applied, all except 1 workflow have some resiliency.

Each plot where resiliency is zero indicates a set of workflows with the same number of separation of duty constraints whose security policy prevents completion. For example, 2 out of 100 workflows with 21 constraints are unable to complete, whilst 36 out of 100 are unable to complete with 30 constraints. No workflow is resilient once 40 constraints have been applied, however some workflows do exist which have some resiliency even with up to 39 out of a possible 45 constraints. The results indicate that some separation of duty constraints can be added or removed with no effect on resiliency.

Computation Time The overall time to compute the resiliency of a workflow can be separated into the time it takes to construct the PRISM model from the workflow definition (*build time*), and the time it takes to verify the finishing property holds in model (*computation time*). If a change is made to the definition before verification, PRISM automatically rebuilds and verifies the model so the total time must be considered. However, once a model has been built and no changes are made, it need not be rebuilt. This is useful where cached, pre-built models can be imagined meaning only computation time need be taken into account when making runtime assignments. It is this time we are interested in improving.

Figure 5 shows how in general, the computation time increases and then decreases despite an incremental introduction of separation of duty constraints. The actual times measured are of course somewhat dependent on the efficiency of the model checker used, in this case PRISM. The maximum average computation time is 2.24 seconds with 11~15 separation of duty constraints. With zero constraints and 41~45 constraints the average computation time is 0.11 and 0.04 seconds respectively. The latter results can intuitively be attributed to the average 0% resiliency value when all 45 constraints are applied. However, even with 26~30 constraints and an average 34.85% resiliency, the average computation time is lower at 0.52 seconds than the time with 11~15 constraints. This would indicate the workflows are on average at their most complex in terms of longest resiliency computation time when approximately 11~15 separation of duty constraints have been applied.

By observing the size of the model that PRISM must solve, in terms of the number of states and transitions, the computation time can be put into context. The maximum average of 2.24 seconds is the computation time taken by PRISM to solve a model with an average 600287 states and 4.75 million transitions. These two values are the maximum average values recorded for states and transitions respectively. As one would expect, computation time appears to be closely related to the size of the model meaning in order to reduce computation time we must look to reduce the size of the model without losing resiliency. The results do indicate that in some cases separation of duty constraints can be added or removed to a workflow without any loss of resiliency.

6 Reducing Computation Time

In this section we provide a methodology to calculate a set of *dummy* security policy constraints (e.g., *redundant* separation-of-duty constraints or removing unused user-task permissions), in order to reduce the resiliency computation time while maintaining the actual resiliency value.

It was shown in Section 5 that in some cases, separation of duty constraints could be added to or removed from a workflow security policy. We are not in a position to say which constraints can be removed as this may weaken the security policy. Therefore we only consider strengthening the policy, in other words adding separation of duty constraints and removing user-task permissions

Table 2. Average computation times and resiliency values when adding a single separation of duty constraint or removing a single permission from the running example policy p_1

	p_1	$+(t_2, t_3)$	$+(t_2, t_5)$	$+(t_3, t_5)$	$+(t_1, t_4)$	$-(u_4, t_4)$	$-(u_4, t_5)$	$-(u_2, t_4)$	$-(u_1, t_1)$
Resiliency (%)	51.16	47.89	51.16	51.16	51.16	39.47	51.16	51.16	51.16
Computation (s)	0.11	0.109	0.141	0.11	0.063	0.047	0.121	0.11	0.062

which in effect can be removed at a later stage if necessary without any loss of security.

6.1 Adding Separations of Duty

In our running example workflow w_1 coming with probabilistic user availabilities (Table 4, Appendix A.4), the resiliency is computed to be 51.16% at an average computation time of 0.11 seconds, based on the average of 50 resiliency calculations. Imagine we now add a new separation of duty constraint (t_2, t_3) to give a new policy $p_2 = (P_2, S_2, B_2)$ where $P_2 = P_1$, $S_2 = S_1 \cup \{(t_2, t_3)\}$, and $B_2 = B_1$. The resiliency of w_1 coming with p_2 is now computed to be 47.89% at an average computation time of 0.109 seconds. In other words, the computation time has reduced by 0.001 seconds but with a loss of 3.27% resiliency.

Now consider adding in turn some alternative separation of duty constraints (t_2, t_5) , (t_3, t_5) and (t_1, t_4) to p_1 to give new policies p_3 , p_4 and p_5 respectively. The resiliency values and average computation times are given in Table 2 where $+(t, t')$ denotes the addition of a separation of duty constraint to p_1 , whilst $-(u, t)$ denotes the removal of a user-task permission from p_1 . The addition of (t_2, t_5) to p_1 (p_3) results in no loss to resiliency but increases the average computation time by 0.031 seconds. Adding (t_3, t_5) to p_1 (p_4) results in no loss to resiliency nor any reduction of average computation time. However, adding (t_1, t_4) to p_1 (p_5) results in no loss to resiliency yet a reduction to the average computation time of 0.047 seconds.

6.2 Removing User Permissions

Similarly we now consider removing a user-task permission (u_4, t_4) to give a new policy $p_6 = (P_6, S_6, B_6)$ where $P_6 = P_1 \setminus \{(u_4, t_4)\}$, $S_6 = S_1$, and $B_6 = B_1$. The resiliency of w_1 coming with p_6 is now computed to be 39.47% at an average computation time of 0.047 seconds. In other words, the computation time has reduced by 0.063 seconds but with a loss of 11.69% resiliency.

We now consider removing in turn some alternative user-task permissions (u_4, t_5) , (u_2, t_4) and (u_1, t_1) from p_1 to give new policies p_7 , p_8 and p_9 respectively. The resiliency values and average computation times are given in Table 2. The removal of (u_4, t_5) from p_1 (p_7) results in no loss to resiliency but increases the average computation time by 0.011 seconds to 0.121 seconds. Removing

Table 3. Average computation times and resiliency values when adding separation of duty constraints or removing permissions from w_B

	w_B	$+s_1$	$+s_2$	$+s_3$	$-p_1$	$-p_2$	$-p_3$
Resiliency (%)	63.96	63.42	62.84	62.21	62.52	60.54	57.92
Computation (s)	6.53	4.28	3.55	3.29	5.09	3.83	1.76

(u_2, t_4) from p_1 (p_8) results in no loss to resiliency nor any reduction of average computation time. However, removing (u_1, t_1) from p_1 (p_9) results in no loss to resiliency yet reduces the average computation time by 0.048 seconds to 0.062 seconds. These results indicate that a selective addition of separation of duty constraints, or removal of user-task permissions can reduce the resiliency computation time without any loss to the actual resiliency value.

6.3 Calculating Dummy Constraints

With the aid of a larger workflow example, we provide a method of calculating an optimal set of *dummy* security policy constraints that minimises resiliency computation time without any reduction to the resiliency value. For clarity we calculate two optimal sets, one of redundant separation of duty constraints that can be added to the policy, and one of user-task permissions that can be removed. Our method could easily be modified to calculate a single set of optimal dummy constraints composed of separation and binding of duty constraints, and user-task permissions.

We consider a single base workflow w_B with 10 tasks and 5 users, coming with a randomly selected security policy p_B composed of 15 separation of duty constraints, 0 binding of duty constraints, and permissions for each task of up to 4 users (29 permissions in total). We use a dynamic user availability model such that each user has an availability for each task of between 0.8 and 1.0 (Table 5, Appendix A.4). The resiliency values and computation times of w_B and all forthcoming variations of it are analysed 50 times and the average values taken. We also use the same computing platform and PRISM model checker set-up as described in Section 5.1. The resiliency of w_B is calculated as 63.96% with an average computation time of 6.53 seconds.

Separations of Duty A test program has been implemented which, given a base workflow, e.g. w_B , calculates monotonically all possible separation of duty constraint combinations that can be added to the workflow security policy. All combinations include only constraints not already included in the base workflow’s security policy. In the case of w_B , the maximum number of constraints is 45 meaning up to 30 can be added. All possible combinations of between 1 and 30 constraints are therefore computed. For each of these a PRISM definition file is automatically generated and analysed by the PRISM model checker. Results are logged for resiliency value, computation time and the set of constraints added to w_B .

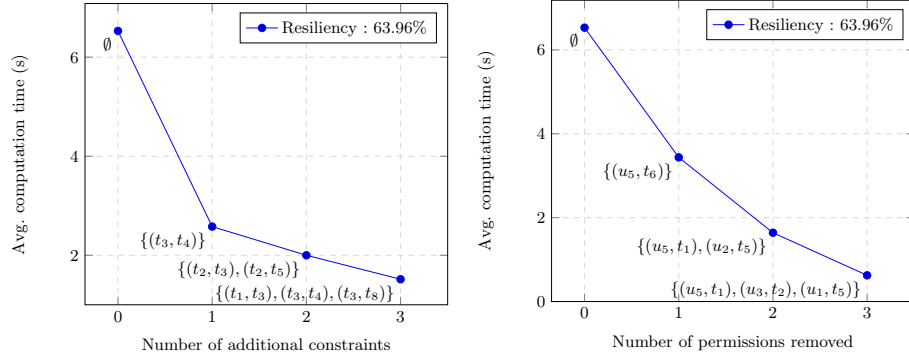


Fig. 6. Impact to resiliency computation time of adding dummy constraints to w_B **Fig. 7.** Impact to resiliency computation time of removing permissions from w_B

The average results of this analysis step are given in Table 3 where the values given for $+s_i$ indicate the average resiliency and computation time for i separation of duty constraints added to w_B . Similarly, values for $-p_i$ indicate the average resiliency and computation time for i user-task permissions removed from w_B . For clarity we only show the impact on computation time of up to 3 additional separation of duty constraints and the removal of up to 3 permissions. In general, adding arbitrary separation of duty constraints in a monotonic fashion is shown to reduce the resiliency computation time but this comes with a reduction in resiliency.

Finding a set of dummy constraints that reduces computation time without reducing resiliency value is found from performing an automatic double sort on the results, first by resiliency value (largest to smallest) and then by time (smallest to largest). The set of dummy constraints that does not change resiliency yet gives the lowest computation time for each i additional constraints is shown in Figure 6. For example, adding the constraint (t_3, t_4) has on average the minimum computation time for one change, that of 2.58 seconds. Notice for three additional constraints, adding $\{(t_1, t_3), (t_3, t_4), (t_3, t_8)\}$ achieves the minimal computation time of 1.52 seconds, thus reducing the original computation time for w_B by 5.01 seconds without lowering its resiliency value.

User-Task Permissions Similarly to adding separation of duty constraints, the results in Table 3 show in general removing arbitrary user-task permissions in a monotonic fashion reduces the resiliency computation time but with a reduction in resiliency. The set of removable permissions shown to give the lowest computation time for each i permissions removed is given in Figure 7. For example, removing the permission (u_5, t_6) has on average the minimum computation time for one change, that of 3.44 seconds. Notice for three permissions, removing $\{(u_5, t_1), (u_3, t_2), (u_1, t_5)\}$ achieves the minimal computation time of 0.63 seconds, thus reducing the original computation time by 5.90 seconds without lowering the resiliency value.

7 Conclusion

We have shown that the way a workflow security policy is designed has a clear impact on the time required to compute the workflow resiliency, which might need to be done at runtime before the execution of each task, in order to ensure that the user-task assignment is suitable. Our results rely on a systematic encoding of a workflow as a probabilistic model and use the ability of the model checker PRISM to efficiently compute resiliency.

We consider this process to be useful in two settings, firstly the workflow design process allowing domain and security experts to assess how resiliency computation time would be impacted following restrictive and unrestrictive changes to the security policy. Secondly, we have proposed an approach adding dummy or artificial security constraints, in order to reduce the computation time. The gain in time can be significant, for instance in our experiment reducing the computation time from 6.53 seconds to 0.63 seconds.

Our experimental results are based on a synthetic workflow, and although it is inspired by real workflows, nothing guarantees that the same efficiency can be gained for all workflows. Hence, we believe our main contribution is the methodology to encode a workflow and to automatically assess its resiliency, based on one of the major probabilistic model-checkers. In the end, the trade-off between resiliency and efficiency can only be resolved by the workflow designer/executer, and we believe that our approach can be helpful in that direction.

In terms of future work we aim to introduce more complex security constraints including cardinality, restricting the number of times a user can be assigned a specific task, to assess their impact on resiliency computation time. We also plan to consider more complex workflows, for instance including loops and choice tasks. These features tend to introduce another level of non-determinism in the execution of the workflow itself, and present as such some challenging aspects in their analysis.

References

1. Workflow handbook 1997. chapter The Workflow Reference Model, pages 243–293. John Wiley & Sons, Inc., New York, NY, USA, 1997.
2. A. Armando and S. E. Ponta. Model checking authorization requirements in business processes. *Computers & Security*, 40(0):1 – 22, 2014.
3. S. Ayed, N. Cuppens-Boulahia, and F. Cuppens. Deploying security policy in intra and inter workflow management systems. In *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, pages 58–65, March 2009.
4. H. E. Bakkali. Enhancing workflow systems resiliency by using delegation and priority concepts. *Journal of Digital Information Management*, 11(4):267 – 276, 2013.
5. D. Basin, S. J. Burri, and G. Karjoth. Obstruction-free authorization enforcement: Aligning security with business objectives. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium, CSF '11*, pages 99–113, Washington, DC, USA, 2011. IEEE Computer Society.

6. D. Basin, S. J. Burri, and G. Karjoth. Optimal workflow-aware authorizations. In *Proceedings of, SACMAT '12*, pages 93–102, New York, NY, USA, 2012. ACM.
7. A. Basu and A. Kumar. Research commentary: Workflow management issues in e-business. *Info. Sys. Research*, 13(1):1–14, Mar. 2002.
8. E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Trans. Inf. Syst. Secur.*, 2(1):65–104, Feb. 1999.
9. R. Botha and J. H. P. Eloff. Separation of duties for access control enforcement in workflow environments. *IBM Systems Journal*, 40(3):666–682, 2001.
10. R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, 55(9):69–77, 2012.
11. R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS management and optimisation in service-based systems. *IEEE Transactions on Software Engineering*, 37(3):387–409, 2011.
12. J. Crampton, G. Gutin, and A. Yeo. On the parameterized complexity and kernelization of the workflow satisfiability problem. *ACM Trans. Inf. Syst. Secur.*, 16(1):4, 2013.
13. J. Crampton and H. Khambhammettu. Delegation and satisfiability in workflow systems. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 31–40. ACM, 2008.
14. D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and parallel Databases*, 3(2):119–153, 1995.
15. L. He, C. Huang, K. Duan, K. Li, H. Chen, J. Sun, and S. A. Jarvis. Modeling and analyzing the impact of authorization on workflow executions. *Future Generation Computer Systems*, 28(8):1177–1193, 2012.
16. L. Herbert and R. Sharp. Precise quantitative analysis of probabilistic business process model and notation workflows. *Journal of Computing and Information Science in Engineering*, 13(1):011007, 2013.
17. H. Hiden, S. Woodman, P. Watson, and J. Cala. Developing cloud applications using the e-science central platform. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1983):20120085, 2013.
18. R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.
19. M. Kohler, C. Liesegang, and A. Schaad. Classification model for access control constraints. In *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE International*, pages 410–417, April 2007.
20. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
21. M. Lowalekar, R. Tiwari, and K. Karlapalem. Security policy satisfiability and failure resilience in workflows. In *The Future of Identity in the Information Society*, volume 298, pages 197–210. Springer Berlin Heidelberg, 2009.
22. J. C. Mace, C. Morisset, and A. van Moorsel. Quantitative workflow resiliency. In *Computer Security - ESORICS 2014*, volume 8712 of *Lecture Notes in Computer Science*, pages 344–361. Springer International Publishing, 2014.
23. J. C. Mace, C. Morisset, and A. van Moorsel. Modelling user availability in workflow resiliency analysis. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security, HotSoS '15*, pages 7:1–7:10, New York, NY, USA, 2015. ACM.

24. F. Martinelli and C. Morisset. Quantitative access control with partially-observable markov decision processes. In *Proceedings of, CODASPY '12*, pages 169–180, New York, NY, USA, 2012. ACM.
25. K. M. Unertl, K. B. Johnson, and N. M. Lorenzi. Health information exchange technology on the front lines of healthcare: workflow factors and patterns of use. *Journal of the American Medical Informatics Association*, 19(3):392–400, 2012.
26. J. Wainer, P. Barthelmess, and A. Kumar. W-rbac - a workflow security model incorporating controlled overriding of constraints. *International Journal of Cooperative Information Systems*, 12:2003, 2003.
27. Q. Wang and N. Li. Satisfiability and resiliency in workflow authorization systems. *ACM Trans. Inf. Syst. Secur.*, 13(4):40:1–40:35, Dec. 2010.

A Workflow Prism Model

Here we provide an overview of the PRISM modelling language and give an encoding of the workflow assignment process defined in Section 4. We also show how PRISM can convert this model definition into an MDP and solve it using its own property specification language to indicate the resiliency of a workflow.

A.1 PRISM Modelling Language

A PRISM model definition constitutes a number of interactive modules that contain one or more local *variables* and *commands* as follows:

```

module name
    variables
    commands
endmodule

```

The values of a module’s local variables define the module’s state while the values of all variables across every module determines the global state of the model. Each variable is defined with a *name*, a *type* restricted to a finite range of integers or a Boolean, and an initial *value*:

```

| name : type init value

```

The initial state for the model is therefore defined by the initial values of all variables. The behaviour of a module is described through a set of local commands. Each command contains a *guard* and one or more *updates*, taking the form:

```

| [label] guard  $\rightarrow$  update1 & ... & updaten;

```

A guard is a predicate over both local variables and those contained in other modules. If a command’s guard equates to true, the guard’s corresponding updates take place assigning one or more local variables with new values. Updating variables is equivalent to a state transition causing the model to move from its current state to a new state. Labelling commands across modules with a common label allows those commands to be synchronised such that a transition consists

of all these commands operating in parallel. Such a transition will only occur when the guards of all its constituent commands equate to true. A guard of the form *true* is an empty guard that always equates to true; commands with such a guard are used to update local variables regardless of their current value. These commands can still be synchronised with others through labelling. A module containing several commands whose guards all equate to true will make a nondeterministic choice over which command to perform.

Named expressions or *formulas* can be included in a PRISM model to avoid the duplication of code and reduce the state space. Essentially a formula's expression can be substituted by its name where ever the expression would be expected.

```
| formula name = expression;
```

A.2 Workflow Encoding

We now provide a PRISM encoding of the assignment process described in Section 4 using the running example workflow in Section 3. The process is driven by the module `main` where the Boolean variables `nt` and `pc` used as flags for correct transition ordering, can be read as *ready for next task* and *policy has been checked* respectively. The formula `pol` is true when an assignment (u, t) satisfies the security policy and the variable `fail` indicates whether the workflow has failed due to a policy violation. Note the command labels are used to synchronise with commands in other modules and can be loosely read as: `[s]` next step; `[p]` check policy; `[a]` make assignment; `[e]` end workflow; `[f]` fail workflow.

```
module main
  nt : bool init true;
  pc : bool init false;
  fail : bool init false;

  [s] nt → (nt'=false) & (pc'=false);
  [p] !nt & !pc → (pc'=true);
  [a] !nt & pc & pol → (nt'=true);
  [e] nt → true;
  [f] !nt & pc & !pol & !fail → (fail'=true);
endmodule
```

Tasks The PRISM module `tasks` selects the next workflow task. Due to the variable type restrictions in PRISM we define the set of tasks T as a finite sequence of integers. The workflow tasks t_1, \dots, t_n where $n = |T|$ are represented by the values $1, \dots, n$ while the workflow's termination point is represented by -1. The current task is stored in a variable `t`, initialised as 0:

```
| t : [-1..n] init 0;
```

In order to satisfy the task ordering relation $<$ and record which tasks have been executed, a Boolean variable is defined for each task such that `true` indicates a

task has been executed and **false** that it has not. For any task t_i we define a variable t_i , initialised as **false**:

```
|  $t_i$  : bool init false ;
```

To enable the selection of a task t_i as the next task in a workflow, a command [s] is defined as follows:

```
| [s]  $!t_i \ \& \ t_j \ \& \ \dots \ \& \ t_m \rightarrow (t'=i) \ \& \ (t_i'=\mathbf{true})$ ;
```

To allow a workflow to terminate, two commands [e] and [f] are defined which set t to the termination point -1 as follows:

```
| [e]  $t_n \rightarrow (t'=-1)$ ;  
| [f] true  $\rightarrow (t'=-1)$ ;
```

The command [e] executes only if t is the last task in the workflow t_n indicating the workflow has ended successfully with all tasks performed. The command [f] executes only if the synchronised command [f] in the **main** module equates to **true**, indicating the workflow has failed and terminated early without all tasks performed. In our running example the module **tasks** is defined as follows:

```
module tasks  
   $t$  :  $[-1..5]$  init 0;  
   $t1$  : bool init false ;  
  ...  
   $t5$  : bool init false ;  
  
  [s]  $!t1 \rightarrow (t'=1) \ \& \ (t1'=\mathbf{true})$ ;  
  [s]  $t1 \ \& \ !t2 \rightarrow (t'=2) \ \& \ (t2'=\mathbf{true})$ ;  
  [s]  $t1 \ \& \ !t4 \rightarrow (t'=4) \ \& \ (t4'=\mathbf{true})$ ;  
  [s]  $t2 \ \& \ !t3 \rightarrow (t'=3) \ \& \ (t3'=\mathbf{true})$ ;  
  [s]  $t3 \ \& \ t4 \ \& \ !t5 \rightarrow (t'=5) \ \& \ (t5'=\mathbf{true})$ ;  
  [e]  $t5 \rightarrow (t'=-1)$ ;  
  [f] true  $\rightarrow (t'=-1)$ ;  
endmodule
```

Users The PRISM module **candidate** selects a user for the next task. We define the set of users U as a finite sequence of integers $1, \dots, n$ where $n = |U|$. The user selected is assigned to a variable pu read as *perspective user*, initialised as 0:

```
|  $pu$  :  $[0..n]$  init 0;
```

For simplicity we do not consider a 'smart' approach to selecting the optimal user for a user-task assignment but instead leave this to PRISM to solve for us, given the correct properties to check. Instead we allow any $u \in U$ to be selected and assigned to pu . Adding a command with an empty guard (**true**) for any user u_i means the update of pu is nondeterministic, as follows:

```
| [s] true  $\rightarrow (pu'=i)$ ;
```

In our running example the **candidate** module as follows:

```

module candidate
  pu : [0..4] init 0;

  [s] true → (pu'=1);
  [s] true → (pu'=2);
  [s] true → (pu'=3);
  [s] true → (pu'=4);
endmodule

```

Permissions Modelling permissions in PRISM is carried out by defining a separate *formula* to capture the user-task permissions for a task. Given a task t_i and a permissions policy P , the user-task permissions for task t_i are captured in a permissions formula p_i as follows:

| **formula** $p_i = t=i \ \& \ (pu=j \mid \dots \mid pu=k);$

Each user u in the conjunction $pu=u_j \mid \dots \mid pu=u_k$ is defined in a permission $(u, t_i) \in P$ such that $u \in up(t_i)$. The permission p_i is therefore *true* if t is the current task (i.e., $t=i$) and the selected user $pu \in up(t_i)$. Evaluating the entire permissions policy P is encapsulated by the formula **perms** which is a conjunction of all permission formulas p_1, \dots, p_k , equating to *true* when no permission has been violated:

| **formula** $perms = p_j \mid \dots \mid p_k;$

Constraints Each separation of duty constraint $(t, t') \in S$ is defined in a distinct PRISM module s_i . Placing each separation of duty constraint in a distinct module allows for easy addition of new constraints and removal of current ones. Evaluating whether a separation (t, t') is satisfied makes it necessary to record the user who executes the first task of the pair (either t or t'). When the second task is to be performed it is possible to verify the user being assigned to the second task is different from the one who performed the first. Note it is not necessary to record users performing any task that does not have a separation (or binding) of duty with any other task.

Given a separation of duty pair $(t, t') \in S$, the user who executes the first task in the pair is recorded in a variable us_i read as the *user of separation of duty i*, initialised as 0:

| $us_i : [0..n]$ **init** 0;

The variable us_i can potentially be assigned any $u \in U$ so an integer sequence $1, \dots, n$ is used where $n = |U|$. Recording whether the separation constraint has been violated is recorded by a Boolean variable fs_i read as *failed separation of duty i* which is *false* if s_i is satisfied or *true* if s_i has been violated:

| $fs_i : \mathbf{bool}$ **init** **false**;

When evaluating a separation of duty constraint, three commands labelled $[p]$ are defined such that only one will execute:

```

| [p] (t=j | t=k) & usi=0 →(usi'=pu);
| [p] (t=j | t=k) & usi!=0 & pu=usi →(fsi'=true);
| [p] (t!=j & t!=k) | (usi!=0 & pu!=usi) →true;

```

Given $(t_i, t_j) \in S$, the first command records the user **pu** when **t** is t_i or t_j and neither task has been performed. The second command records the constraint has been violated when **pu** is the same as us_i and the third command does nothing when the constraint is satisfied or **t** is neither t_i or t_j . In our running example, the constraint $(t_2, t_4) \in S_1$ is defined as:

```

| module s1
|   us1 : [0..4] init 0;
|   fs1 : bool init false;
|
|   [p] (t=2 | t=4) & us1=0 →(us1'=pu);
|   [p] (t=2 | t=4) & us1!=0 & pu=us1 →(fs1'=true);
|   [p] (t!=2 & t!=4) | (us1!=0 & pu!=us1) →true;
| endmodule

```

Checking the entire separations of duty policy S is encapsulated by the formula **sods** which is a conjunction of all fs_1, \dots, fs_n variables defined in modules s_1, \dots, s_n :

```

| formula sods = !fs1 & ... & !fsn;

```

The formula **sods** equates to true when no separation of duty constraint has been violated, in other words all variables fs_1, \dots, fs_n are **false**. Similarly, each binding of duty constraint $(t, t') \in B$ is defined in a distinct PRISM module b_i :

```

| module bi
|   ubi : [0..n] init 0;
|   fbi : bool init false;
|
|   [p] (t=j | t=k) & ubi=0 →(ubi'=pu);
|   [p] (t=j | t=k) & ubi!=0 & pu!=ubi →(fbi'=true);
|   [p] (t!=j & t!=k) | (ubi!=0 & pu=ubi) →true;
| endmodule

```

Note the Boolean variable fb_i read as *failed binding of duty i is true* if b_i has been violated, in other words the user for t_j and t_k is different. The formula **bods** encapsulates the binding of duty policy B for all modules b_1, \dots, b_n , equating to *true* when no binding of duty has been violated, in other words all variables fb_1, \dots, fb_n are *false*:

```

| formula bods = !fb1 & ... & !fbn;

```

Policy Having defined the formulas **perms**, **sods** and **bods** we are able to evaluate the workflow security policy $p = (P, S, B)$ by defining a formula **pol**:

| **formula** *pol* = perms & sods & bods;

The policy formula *pol* equates to *true* only if a given user-task assignment satisfies the permissions, separation and binding of duties constraints. Note how *pol* is used by the **main** module as a guard in the [a] and [f] commands to evaluate whether an assignment can be made or the workflow has failed respectively.

Probabilistic Availability In order to calculate a workflow’s resiliency we extend our PRISM model to include a probabilistic user unavailability model. PRISM allows probabilistic variable updates with commands of the form:

| [*label*] *guard* $\rightarrow prob_1 : update_1 + \dots + prob_n : update_n$;

This states the probability of *update*₁ is *prob*₁ and so forth such that the sum of *prob*₁ to *prob*_{*n*} is 1. Only one of the updates will take place with its given probability assuming the guard is true. For example, a Boolean variable **available** has a probability of 0.75 to be true and 0.25 to be false with a command of the form:

| [*label*] *guard* $\rightarrow 0.75 : (\text{available}' = \text{true}) + 0.25 : (\text{available}' = \text{false})$;

Full details of encoding user availability are given in [?].

A.3 Model Building and Verification

The property we wish to verify is defined in PRISM as:

| Pmax=? [F (t=-1) & (!fail)]

where: Pmax=? asks PRISM to find the maximal probability of the formula in between brackets to be true; the operator F is the “Eventually” operator, i.e., F *p* is true if the statement *p* eventually holds in the system. In this case, we ask PRISM to find the maximal property so that, eventually, the termination point has been reached, (t=-1) and the workflow has not failed, (! fail).

In order to verify the satisfiability property, PRISM first builds the corresponding model by converting a given definition into an MDP. To do this PRISM computes the set of all reachable states from the initial state of the model and the transition matrix which represents the model. The transition matrix contains probabilities for all state transitions in the model. A non-zero entry represents the probability of going from state *i* to state *j*, row *i* to column *j* in the matrix. Verification of the MDP is based on exhaustive search and numerical solution. As we are solving the MDP to find the maximum probability we do not require a reward function. The numerical result returned by PRISM is the value calculated in the initial state of the MDP.

A.4 Probabilities for User Availability

Table 4. User probabilistic availabilities used to compute resiliency for running example workflow w_1 in Sections 4, 6.1 and 6.2

	t_1	t_2	t_3	t_4	t_5
u_1	0.9568	0.8338	0.7206	0.7231	0.7099
u_2	0.8565	0.9210	0.8016	0.8091	0.9460
u_3	0.8263	0.8617	0.7705	0.7192	0.7117
u_4	0.7238	0.8999	0.9486	0.8413	0.8063

Table 5. User probabilistic availabilities used to compute resiliency when assessing policy changes (Section 5) and for the base workflow w_B (Section 6.3)

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
u_1	0.9297	0.9996	0.8506	0.8737	0.9057	0.8365	0.9514	0.8555	0.9665	0.9875
u_2	0.8381	0.8883	0.8231	0.8726	0.8099	0.9732	0.9852	0.8506	0.9825	0.8089
u_3	0.9653	0.9246	0.8429	0.9491	0.9597	0.8394	0.8560	0.9585	0.8304	0.8330
u_4	0.9263	0.9691	0.8241	0.9932	0.9868	0.9792	0.9162	0.9339	0.9868	0.8049
u_5	0.9724	0.8817	0.9401	0.8261	0.9339	0.8432	0.9329	0.8682	0.8231	0.8842